Studi Pengaruh Ukuran Blok Terhadap Kinerja Algoritma Block A* pada Peta Statis

Dody Dharma[1], Rinaldi Munir[2], Mamat Rahmat[3]
Kelompok Keahlian Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Indonesia
[1]dody@stei.itb.ac.id, [2]rinaldi-m@stei.itb.ac.id, [3]13512007@std.stei.itb.ac.id

***Abstrak*— Algoritma Block A* adalah pengembangan algoritma A* pada grid. Algoritma ini membagi peta berupa grid menjadi blok-blok berukuran sama. Pra-komputasi dilakukan terhadap setiap kemungkinan konfigurasi halangan blok untuk memperoleh data jarak terpendek lokal antar sisi sel pada blok. Data ini disimpan ke dalam struktur data bernama LDDB dan digunakan untuk mempercepat perhitungan ekspansi blok pada algoritma utama Block A*. Peningkatan ukuran blok berpotensi meningkatkan kinerja waktu pemrosesan algorima Block A*. Hasil pengujian menunjukkan bahwa ukuran blok mempengaruhi peningkatan kinerja waktu pemrosesan algoritma Block A*. Terjadi peningkatan kinerja secara logaritmik dimana secara umum terdapat penurunan waktu proses yang signifikan pada perbesaran ukuran blok untuk ukuran blok yang kecil. Namun, peningkatan semakin tidak signifikan saat ukuran blok diperbesar. Pada ukuran blok yang cukup besar, secara umum terjadi penurunan kinerja akibat dari proses inisialisasi yang lebih dominan pada ukuran blok yang besar.**

Kata Kunci— pathfinding, graf, algoritma A*, algoritma Block A*

I. PENDAHULUAN

Algoritma *pathfinding* umum digunakan dalam permainan komputer. Algoritma *pathfinding* adalah pencarian lintasan terpendek yang dapat dilalui *agent* dari titik awal ke titik tujuan pada suatu peta. Dalam permainan komputer, algoritma *pathfinding* tidak hanya dijalankan sekali, tetapi umumnya dijalankan berkali-kali selama permainan untuk titik awal dan titik tujuan yang berbeda. Dalam permainaan dengan banyak *agent*, beberapa proses *pathfinding* perlu dilakukan secara simultan. Hal ini perlu dilakukan secara *real-time* sehingga dibutuhkan solusi algoritma *pathfinding* yang efisien agar lintasan terpendek dapat dihitung dengan cepat menggunakan sumber daya prosesor dan memori yang terbatas [1].

Algoritma *pathfinding* yang populer digunakan dalam permainan komputer adalah algoritma A*. Algoritma Block A* adalah pengembangan dari algoritma A* pada graf berbasis *grid* dengan memperkenalkan struktur data baru bernama *Local Distance Database* (LDDB). Pada algoritma Block A*, *grid* dibagi ke dalam blok-blok berukuran sama. LDDB menyimpan data panjang lintasan terpendek antar setiap pasang sel pada sisi blok. Berbeda dengan algoritma A* yang melakukan ekspansi pada satuan sel, algoritma Block A* melakukan ekspansi pada satuan blok. Setiap langkah ekspansi memanfaatkan data dari LDDB untuk mempercepat perhitungan [2].

Sebelum algoritma Block A* dijalankan, perlu dilakukan pra-komputasi LDDB terlebih dahulu menggunakan algoritma *Breadth First Search*. LDDB menyimpan data untuk setiap kemungkinan konfigurasi halangan pada blok. Kompleksitas ruang dari struktur data LDDB adalah $O(2^{(m*n)} * (m + n)^2)$. Hal ini berarti besar data yang disimpan oleh LDDB meningkat secara eksponensial seiring dengan meningkatnya ukuran blok. Karena keterbatasan memori pada komputer, ukuran blok yang dapat ditangani oleh LDDB menjadi sangat terbatas yaitu hingga ukuran 5 x 5. Algoritma Block A* didesain untuk menangani peta yang dinamis sehingga setiap kemungkinan blok dihitung pada pra-proses untuk menangani adanya perubahan pada peta meskipun terdapat blok-blok yang tidak mungkin muncul dalam peta [2].

Dalam kasus peta statis, blok yang diperlukan hanyalah yang terdapat pada peta asal saja. Dengan hanya melakukan pra-komputasi terhadap blok pada peta asal, kompleksitas ruang dari LDDB dapat direduksi menjadi polinomial terhadap ukuran peta sehingga dimungkinkan untuk menangani ukuran blok yang besar. Didasarkan dari hasil eksperimen [2] dimana disimpulkan bahwa blok ukuran 5x5, Block A* memiliki kinerja waktu 2x lebih baik dibanding A*, peningkatan ukuran blok LDDB memiliki potensi meningkatkan kinerja waktu pemrosesan algoritma Block A*.

II. PENELITIAN TERKAIT

Sama seperti algoritma A*, kinerja algoritma Block A* bergantung kepada fungsi heuristik yang digunakan. Jika fungsi heuristik yang digunakan buruk, maka kinerja pencarian lintasan akan menurun bahkan bisa tidak mendapatkan hasil. Jika fungsi heuristik yang digunakan adalah fungsi heuristik yang baik, maka kinerja Block A* pun meningkat.

Optimasi algoritma Block A* dilakukan dengan menggunakan fungsi heuristik bernama diferensial terkompresi. Fungsi heuristik ini menggunakan data yang diolah sebelumnya sebagai basis heuristiknya untuk meningkatkan kualitas pencarian. Fungsi heuristik diferensial menyimpan jarak terpendek dari suatu titik ke titik lainnya pada area pencarian. Data ini akan memberikan batasan jarak dalam pencarian lintasan. Dengan optimasi ini, jumlah memori yang digunakan dapat dikurangi menjadi 75%, tetapi dengan sedikit penurunan kinerja waktu sebesar 3,8% [3].

III. KONSEP TERKAIT

A. Teori Graf

Graf terdiri dari dua komponen, yaitu himpunan tidak kosong dari *node* (*V*) dan himpunan dari *edge* (*E*) dimana sebuah *edge* menghubungkan sepasang *node*. *Edge* pada graf dapat memiliki arah dan bobot. Berdasarkan orientasi arahnya, graf terbagi ke dalam 2 jenis. Pada graf berarah, *edge* (*u*,*v*) berarti *node v* terhubung langsung dari *node u*. Hal ini disebut juga *node v* tetangga dari *node u*. Pada graf tidak berarah, *edge* (*u*,*v*) berarti *node v* terhubung langung dari *u* dan *node u* terhubung langung dari *node v*. Atau dengan kata lain, *node u* tetangga dari *node v* dan *node v* tetangga dari *node v*. Berdasarkan bobotnya, graf terbagi ke dalam 2 jenis. Pada graf berbobot, setiap *edge* (*u*,*v*) masing-masing memiliki bobot *w*(*u*,*v*). Pada graf tidak berbobot, setiap *edge* (*u*,*v*) dianggap memiliki bobot yang sama atau umumnya dianggap memiliki bobot 1 satuan [4].

Lintasan atau *path* adalah barisan *edge* yang dimulai dari suatu *node*, berarkhir pada suatu *node* dan berurutan dari *node* ke *node* melalui *edge*. Pada graf berbobot, panjang lintasan adalah total bobot dari *edge-edge* pada lintasan. Pada graf tidak berbobot, panjang lintasan adalah banyaknya *edge* pada lintasan. Lintasan terpendek atau *shortest path* antara dua *node* adalah lintasan yang memiliki panjang terkecil diantara semua lintasan lain antar dua *node* tersebut [4].

Peta pada permainan komputer umumnya berupa *grid* dan dapat direpresentasikan ke dalam graf. Setiap sel pada *grid* dapat berupa ruang kosong yang dapat ditelusuri atau berupa halangan. *Node* mewakili sel-sel yang merupakan ruang kosong sedangkan definisi *edge* tergantung dari tipe gerak *agent*. Jika gerak *agent* hanya 4 arah (*tile grid*), maka dua *node* dihubungkan oleh *edge* dengan panjang 1 satuan jika dan hanya jika kedua *node* adalah sel ruang kosong dan bersisian secara vertikal atau horisontal. Untuk gerak *agent* 8 arah (*octile grid*), ada tambahan jenis *edge* dengan bobot $\sqrt{2} \approx 1.414$ satuan jika dan hanya jika kedua *node* adalah sel ruang kosong, bersisian secara diagonal serta dua sel lain yang bersisian dengan keduanya adalah ruang kosong [5].

B. Algoritma Dijkstra

Dijkstra adalah algoritma untuk mencari panjang lintasan terpendek dari suatu *node* ke semua *node* lain pada graf berbobot non-negatif. Algoritma ini juga dapat digunakan untuk mencari panjang lintasan terpendek antar setiap pasang *node* dengan cara menjalankannya pada setiap *node* [6].

```
   Dijkstra(G, weight, s)
1.    for each node v
2.        v.distance = ∞
3.    s.distance = 0
4.    PQ = new PriorityQueue()
5.    PQ.push(s)
6.    while PQ ≠ ∅
7.        u = PQ.extract_min()
8.        for each node v that is adjacent to node u
9.            if v.distance > u.distance + weight(u, v)
10.            v.distance = u.distance + weight(u, v)
11.            if not PQ.contains(v)
12.                PQ.push(v)
```

Algoritma 1. Dijkstra [6]

C. Algoritma A*

Algoritma A* adalah algoritma yang dapat digunakan untuk berbagai masalah pencarian. Algoritma ini memanfaatkan fungsi heuristik pada setiap langkah penelusuran untuk menentukan *node* mana yang menjanjikan untuk ditelusuri selanjutnya. Fungsi heuristik menghitung estimasi panjang lintasan (total bobot) dari suatu *node* ke *node* tujuan [7].

```
1. Add the starting node to the open list.
2. Repeat the following steps:
    a. Look for the node which has the lowest f on the
       open list. Refer to this node as the current
       node.
    b. Switch it to the closed list.
    c. For each reachable node from the current node
        i.   If it is on the closed list, ignore it.
        ii.  If it isn't on the open list, add it to the
             open list. Record the f, g, and h value of
             this node.
        iii. If it is on the open list already, check
             to see if this is a better path. If so,
             recalculate the f and g value.
    d. Stop when
        i.  Add the target node to the closed list.
        ii. Fail to find the target node, and the open
            list is empty.
```

Algoritma 2. A* [7]

A* memilih lintasan yang meminimalkan fungsi $f(n)$

$$f(n) = g(n) + h(n)$$

dimana *n* adalah *node* terakhir pada lintasan, *g*(*n*) adalah panjang lintasan dari *node* awal ke *n* dan *h*(*n*) adalah heuristik yang mengestimasi panjang dari lintasan terpendek dari *n* ke *node* tujuan. Pemanfaatan fungsi heuristik ini menyebabkan algoritma A* lebih efisien dari segi jumlah *node* yang ditelusuri dibandingkan algoritma lain seperti algoritma Dijkstra [7].

Terdapat beberapa fungsi heuristik yang umum digunakan pada *grid* [8]:

1. *Manhattan Distance*
   Heuristik ini cocok digunakan untuk grid dengan gerak agent 4 arah.
   $$h(n) = |n.x - goal.x| + |n.y - goal.y|$$
2. *Diagonal distance*
   Heuristik ini cocok digunakan untuk grid dengan gerak agent 8.
   $$h(n) = max(dx, dy) + (\sqrt{2} - 1) * min(dx, dy)$$
   Dimana *dx* adalah selisih *x* dari sel *n* dan goal, sedangkan *dy* adalah selisih *y* dari sel *n* dan goal.

D. Algoritma Block A*

Algoritma Block A* adalah pengembangan dari algoritma A* pada graf berbasis *grid* dengan memperkenalkan struktur data baru bernama *Local Distance Database* (LDDB). Pada algoritma Block A*, *grid* dibagi ke dalam blok-blok berukuran sama. Berbeda dengan algoritma A* yang melakukan ekspansi pada satuan sel, algoritma Block A* melakukan ekspansi pada satuan blok. Setiap langkah ekspansi memanfaatkan data dari

LDDB untuk mempercepat perhitungan panjang lintasan terpendek dalam sebuah blok [2].

Local Distance Database (LDDB) menyimpan panjang lintasan terpendek antar sel-sel pada sisi blok. Misal *x* dan *y* adalah sel pada sisi blok. Query *LDDB*[*x, y*] mengembalikan panjang lintasan terpendek antara dua sel *x* dan *y*. Terdapat $2^{b \times b}$ konfigurasi halangan untuk blok berukuran *b* x *b*. Untuk setiap konfigurasi halangan pada blok, LDDB menyimpan panjang lintasan optimal setiap pasangan (*x*,*y*) [2].

```
   Expand (currBlock, Y)
1.     for side of curBlock with neighbor nextBlock do
2.         for valid egress node x on current side do
3.             x' = egress neighbor of x on current side
4.             x.g = min_{y∈Y}(y.g + LDDB(y, x), x.g)
5.             x'.g = min(x'.g, x.g + cost(x, x'))
6.         end for
7.         newheapvalue = min_{updated x'}(x'.g + x'.h)
8.         if newheapvalue < nextBlock.heapvalue then
9.             nextBlock.heapvalue = newheapvalue
10.          if nextBlock not in OPEN then
11.              insert nextBlock into OPEN
12.          else
13.              UpdateOPEN(nextBlock)
14.          end if
15.     end if
16. end for

   Block A* (LDDB, start, goal)
1.     startBlock = init(start)
2.     goalBlock = init(goal)
3.     length = ∞
4.     insert startBlock into OPEN
5.     while (OPEN ≠ empty) and
               ((OPEN.top).heapvalue < length) ) do
6.         curBlock = OPEN.pop
7.         Y = set of all curBlock's ingress nodes
8.         if curBlock == goalBlock then
9.             length = min_{y∈Y} (y.g + dist(y, goal),
    length)
10.     end if
11.     Expand(curBlock, Y)
12. end while
13. if length ≠ ∞ then
14.     Reconstruct solution path
15. else
16.     return Failure
17. end if
```

Algoritma 3. Block A* [2]

Algoritma utama Block A* diadaptasi dari algoritma A* untuk melakukan ekspansi blok pada setiap tahap. Mirip dengan algoritma A*, siklus dari algoritma Block A* adalah mengambil elemen berupa blok dari *open list* dan melakukan ekspansi dari blok saat ini ke blok tetangganya. LDDB digunakan untuk memperbarui nilai *g* dari sel pada sisi blok yang sedang diekspansi (Yap dkk., 2011).

Pada Algoritma Block A*, himpunan *ingress cell* adalah himpunan sel-sel kosong pada sisi blok yang memiliki nilai *g* lebih kecil dibanding saat ekspansi sebelumnya, atau bernilai bukan tak hingga jika blok diekspansi pertama kali. Sedangkan *egress cell* adalah himpunan sel kosong pada sisi blok yang bersisian dengan sel kosong di blok lain. Sel kosong yang berada di blok lain tersebut dinamakan *egress neighbour*. *OPEN* adalah *priority queue*.

Prosedur *Expand*(*currBlock, Y*) adalah prosedur untuk melakukan ekspansi dari *currBlock* ke blok tetangga dari *currBlock* dimana *Y* adalah himpunan *ingress cell* dari *currBlock*. Pada setiap sisi blok dengan *nextBlock* adalah tetangga pada sisi tersebut :

- Nilai *g egress cell* pada sisi tersebut diperbarui dengan cara menghitung nilai minimal diantara setiap kemungkinan *ingress*. Nilai yang dibandingkan adalah jumlahan *g ingress cell* dan jarak terpendek dari *ingress cell* ke *egress cell*. Jarak ini diambil dari LDDB.
- Nilai *g egress neighbour* pada sisi tersebut diperbarui dengan cara menghitung nilai minimal diantara setiap kemungkinan *egress cell*-nya. Nilai yang dibandingkan adalah jumlahan *g egress cell* yang nilainya diperbarui pada proses di atas dan panjang *edge* antara *egress cell* dan *egress neighbour*.
- Calon nilai *heapvalue* yang baru untuk *nextBlock* dihitung cara mencari nilai minimal diantara *g egress neighbour* yang nilainya diperbaharui pada proses di atas. Jika calon *heapvalue* lebih kecil dari *heapvalue* yang tercatat pada blok, perbarui nilai *heapvalue* dan masukkan *nextBlock* ke *OPEN*.

Prosedur *BlockA**(*LDDB, start, goal*) mirip dengan algoritma A* namun ekspansi dilakukan pada satuan blok. Mula-mula dilakukan inisialisasi terhadap sel *start* dan sel *goal* dimana masing-masing ditentukan blok dimana mereka berada dan diberi nama *startBlock* dan *goalBlock*. Pada kedua blok tersebut, dihitung jarak terpendek dari *start*/*goal* ke *egress cell* pada masing-masing blok. Jarak yang diperoleh disimpan ke LDDB. Selain itu nilai *g egress cell* pada startBlock diinisialisasi dengan jarak terpedek dari *start* ke *egress cell*. Jarak ini diambil dari LDDB.

Nilai *length* diinisialisasi tak hingga. *startBlock* dimasukkan ke *OPEN*. Selama *OPEN* tidak kosong dan *heapvalue* elemen terkecil dari *OPEN* lebih kecil dari *length*, lakukan hal berikut :

- Keluarkan elemen terkecil sebagai *currBlock*
- Jika *currBlock* sama dengan *goalBlock*, maka calon panjang lintasan terpendek dari *start* ke *goal* dapat diperoleh dengan menghitung nilai minimal dari hasil perhitungan *g ingress* ditambah jarak *ingress cell* ke *goal* untuk setiap kemungkinan *ingress*. Jarak ini diperoleh dari LDDB. Jika nilai diperoleh lebih kecil dari length, perbarui nilai length.
- Lakukan ekspansi currBlock dengan prosedur *Expand*()

Jika pada akhir algoritma, nilai *length* tetap tak hingga, maka tidak ada lintasan dari *node* awal ke *node* tujuan. Jika tidak, maka nilai *length* adalah panjang lintasan optimal.

IV. Analisis dan Perancangan

A. Kendala Penggunaan Memori LDDB Standar

LDDB standar didesain untuk menangani semua kemungkinan perubahan pada peta sehingga setiap

kemungkinan konfigurasi halangan pada blok disimpan pada LDDB. Banyaknya kemungkinan konfigurasi halangan pada blok berukuran k x k ada $2^{(k*k)}$ dan banyaknya pasangan sel pada sisi blok ada $(4k-4)^2$. Sehingga seluruh data panjang yang perlu disimpan pada LDDB adalah sebanyak $2^{(k*k)} * (4k-4)^2$. Kompleksitas ruang dari struktur data ini adalah $O(2^{(k^2)} * (k)^2)$. Kompleksitas ruang yang eksponensial terhadap ukuran blok menyebabkan kebutuhan memori yang tinggi untuk ukuran blok yang besar.

Pada kasus peta statis, peta tidak berubah. Solusi untuk menangani masalah di atas dalam kasus peta statis adalah dengan melakukan pra-komputasi LDDB hanya untuk blok-blok yang terdapat pada peta. Banyaknya blok berukuran k x k dalam peta berukuran R x C adalah $\left\lceil\frac{R}{k}\right\rceil * \left\lceil\frac{C}{k}\right\rceil$. Dalam kasus terburuk, seluruh data panjang yang perlu disimpan adalah sebanyak $\left\lceil\frac{R}{k}\right\rceil * \left\lceil\frac{C}{k}\right\rceil * (4k-4)^2$. Kompleksitas ruang dari struktur data ini adalah $O(R * C)$ atau linier terhadap ukuran *grid*. Dengan solusi tersebut, ukuran blok dapat diperbesar menjadi lebih dari 5 x 5.

B. Perancangan LDDB

Pada peta dengan gerak *agent* 4 arah (*tile grid*), pra-komputasi LDDB dapat dilakukan dengan menggunakan algoritma *Breadth First Search* pada tiap sel karena bobot *edge* yang menghubungkan dua sel adalah sama yaitu 1 satuan. Namun algoritma BFS tidak dapat digunakan untuk graf berbobot. Untuk menangani peta dengan gerak *agent* 8 arah (*octile grid*) dimana terdapat 2 jenis *edge* dengan panjang yang berbeda, dibutuhkan algoritma *all-pair shortest path* yang dapat menangani graf berbobot.

Persoalan *all-pair shortest path problem* dapat diselesaikan menggunakan algoritma Floyd Warshall. Algoritma ini memiliki kompleksitas waktu $O(V^3)$. Cara lain yang dapat dilakukan adalah dengan menjalankan algoritma Dijkstra pada tiap sel dengan kompleksitas $O(V_{awal} * (V+E) * log(V))$. Pada graf berupa *grid*, setiap sel maksimal bertetangga dengan 8 sel lain, sehingga $|E| \leq 8|V|$ dan $O(E) = O(V)$, sehingga *all-pair shortest path* pada *grid* menggunakan algoritma Dijkstra memiliki kompleksitas waktu $O(V_{awal} * V * log(V))$ [6].

Karena algoritma Dijkstra memiliki kompleksitas yang lebih baik dibandingkan algoritma Floyd warshall, maka akan digunakan algoritma Dijkstra dalam pra-komputasi LDDB. Data blok dipra-komputasi dan disimpan hanya untuk blok yang terdapat pada peta saja. Algoritma pra-komputasi solusi LDDB untuk peta statis dapat dilihat pada Algoritma 4.

```
   Precompute(grid, text)
1.    grid.readFromMaps(text)
2.    grid.generateBlocks()
3.    for block in grid.blocks() do
4.        for cell in block.cells() do
5.            if cell.isOnSide() then
6.                distances = block.runDijkstra(cell)
7.                LDDB.update(distances)
8.            endif
9.        endfor
10. endfor
```

Algoritma 3. Pra-komputasi LDDB

Perubahan hanya dilakukan pada struktur data LDDB dan algoritma pra-komputasi LDDB. Algoritma Block A* tidak mengalami perubahan dari algoritma aslinya. Heuristik yang akan digunakan pada impelementasi algoritma Block A* adalah *octile distance*.

C. Analisis Peningkatan Kinerja Waktu Pemrosesan

Dengan menggunakan solusi LDDB untuk peta statis, ukuran blok dapat dibuat lebih besar dari 5 x 5. Menurut Yap dkk. (2011), ukuran blok 4 x 4 dinilai sudah cukup efektif. Bahkan algoritma Block A* untuk blok ukuran 2 x 2 memiliki kinerja waktu pemrosesan rata-rata 2 kali lebih cepat dari algoritma A*. Kinerja ini memiliki potensi untuk dikembangkan lebih lanjut dengan memperbesar ukuran blok. Selanjutnya akan dijelaskan analisis kompleksitas waktu algoritma A* dan Block A*.

Menurut Cui & Shi (2011), dengan pemanfaatkan fungsi heuristik, jumlah *node* yang ditelusuri oleh algoritma A* pada pencarian graf lebih sedikit dibanding jumlah *node* yang ditelusuri oleh algoritma Dijkstra. Kompleksitas waktu algoritma A* dapat dipandang sebagai kompleksitas waktu algoritma Dijkstra dikali perhitungan fungsi heuristik yaitu $O((V+E) * log(V)) * O(perhitungan\ heuristik)$.

Misal ukuran *grid* adalah R x C. Banyaknya *node* pada *grid* tersebut kompleksitasnya adalah $O(V) = O(R * C)$. Pada graf berupa *grid*, setiap sel maksimal bertetangga dengan 8 sel lain, sehingga banyaknya *edge* kompleksitasnya linier terhadap banyaknya *node*, yaitu $O(E) = O(V) = O(R * C)$. Karena kompleksitas perhitungan heuristik $O(1)$, maka kompleksitas waktu dari algoritma A* pada *grid* adalah $O((V+E) * log(V)) * O(1) = O(R * C * log(R * C))$.

Pada algoritma Block A*, ukuran blok yang besar memiliki kelebihan dan kekuarangan. Kelebihannya adalah jumlah *node* (dalam hal ini berupa blok) yang diekspansi menjadi lebih sedikit. Kekurangannya adalah proses ekspansi pada tiap blok yang lebih lama karena ukuran blok yang besar menyebabkan kombinasi *ingress-egress cell* lebih banyak. Selain itu kekurangannya adalah proses *init()* yang menggunakan algoritma Dijkstra pada blok awal dan blok tujuan yang lebih besar dapat memakan waktu yang lebih lama.

Berdasarkan algoritma II.3, kompleksitas algoritma Block A* adalah kompleksitas algoritma A* pada *grid* berisi blok-blok dikali kompleksitas ekspansi ditambah kompleksitas algoritma Dijkstra pada saat pemanggilan fungsi *init()*. Misal ukuran *grid* adalah R x C dan ukuran blok adalah k x k. Banyaknya blok dalam *grid* tersebut adalah $\left\lceil\frac{R}{k}\right\rceil * \left\lceil\frac{C}{k}\right\rceil$. Banyaknya *ingress cell* dan *egress cell* maksimal adalah banyaknya sel pada sisi blok yaitu $4k$ - 4. Ekspansi dilakukan untuk setiap pasang *ingress cell* dan *egress cell* dimana kompleksitas *ingress cell* adalah $O(k)$ dan kompleksitas *egress cell* juga adalah $O(k)$. Sehingga kompleksitas waktu algoritma Block A* adalah :

$$O(A*\ pada\ blok) * O(\text{ekspansi}) + O(\text{dijkstra pada blok})$$

$$= O(\left\lceil\frac{R}{k}\right\rceil * \left\lceil\frac{C}{k}\right\rceil * log\ (\left\lceil\frac{R}{k}\right\rceil * \left\lceil\frac{C}{k}\right\rceil) * k^2 + k^2 * log(k^2))$$

$$= O(R * C * log(\frac{R * C}{k^2}) + k^2 * log(k^2))$$

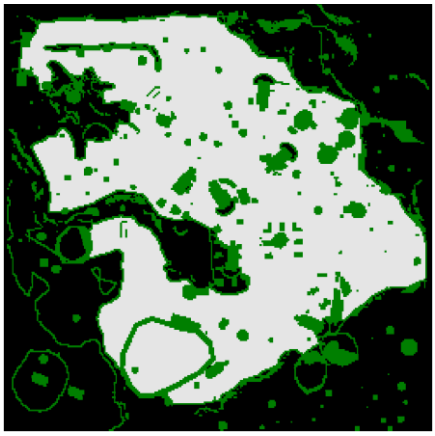
$$= O(\frac{R * C * log(R * C)}{log(k)} + k^2 * 2 * log(k))$$

Dari hasil perhitungan kompleksitas di atas, dapat dianalisis bahwa algoritma Block A* pada *k* yang kecil akan mengalami peningkatan kinerja dengan orde log *k*. Hal ini dikarenakan bagian kiri pada bentuk jumlahan kompleksitas di atas nilainya lebih signifikan dibandingkan bagian kanan. Sedangkan untuk *k* yang besar, bagian kanan akan menjadi lebih signifikan sehingga algoritma Block A* akan mengalami penurunan kinerja dengan orde k^2.

V. Hasil Implementasi dan Pengujian

Implementasi dilakukan dengan bahasa Python. Implementasi struktur data *binary-min-heap* untuk *priority queue* dalam algoritma Block A* dan Dijkstra dilakukan dengan bantuan modul *heapq* dari pustaka standar bahasa pemrograman Python.

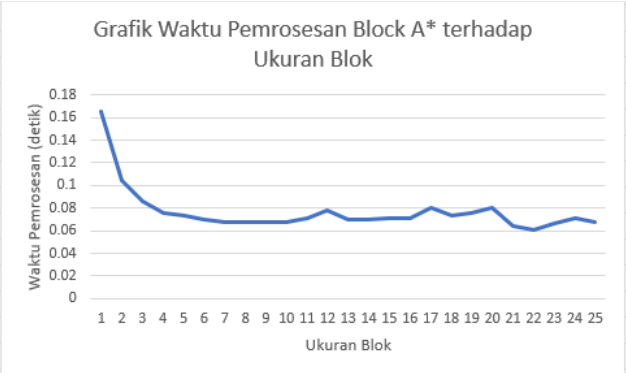
Data yang digunakan untuk pengujian adalah peta permainan Dragon Age: Origins dengan kode brc000d berukuran 261 x 257.



Gambar 1. Peta brc00d pada permainan Dragon Age: Origins

Peta ini diambil dari dataset 2D *Pathfinding Benchmarks* https://movingai.com/benchmarks/grids.html (Sturtevant, N. R, 2012). Dataset tersebut berisi data peta permainan berupa matriks karakter dan 851 buah skenario sel awal dan sel tujuan yang berbeda-beda. Pada masing-masing skenario diberikan nilai panjang rute terpendek.

Berdasarkan hasil pengujian, hasil implementasi algoritma Block A* menghasilkan panjang lintasan yang cukup optimal, yaitu dengan perbedaan kurang dari 5%. Grafik hasil pengujian waktu proses algoritma Block A* dapat dilihat pada Gambar IV.I. Grafik tersebut adalah waktu rata-rata pemrosesan terhadap ukuran blok.



Gambar 2. Hasil pengujian waktu pemrosean rata-rata algoritma Block A*

VI. Kesimpulan

Implementasi struktur data LDDB dan penyesuaian algoritma Block A* pada kasus peta statis untuk menangani ukuran blok yang besar berhasil dilakukan. Pengujian menunjukkan bahwa Block A* mengalami peningkatan kinerja secara logaritmik dimana secara umum terdapat penurunan waktu proses yang signifikan pada perbesaran ukuran blok untuk ukuran blok yang kecil. Namun, peningkatan semakin tidak signifikan saat ukuran blok diperbesar. Pada ukuran blok yang cukup besar, secara umum terjadi penurunan kinerja. Hasil ini sesuai dengan analisis yang telah dikemukakan sebelumnya. Anomali kinerja waktu dimana terjadi kenaikan dan penururan kinerja diakibatkan waktu proses juga bergantung kepada kondisi peta. 3. Block A* lebih baik digunakan dengan ukuran blok LDDB yang kecil. 3.

Daftar Pustaka

[1] Botea, A., Bouzy, B., Buro, M., Bauckhage, C., & Nau, D. (2013). Pathfinding in games. Dagstuhl Follow-Ups (Vol. 6). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[2] Yap, P., Burch, N., Holte, R. C., & Schaeffer, J. (2011). Block A*: Database-Driven Search with Applications in Any-Angle Path-Planning. AAAI.

[3] Wicaksono, T. B., & Munir, R. (2012). Pengembangan Heuristik Diferensial Terkompresi untuk Algoritma Block A*. JUTI: Jurnal Ilmiah Teknologi Informasi, 10(2), 96-103.

[4] Rosen, K. H. (2012). Discrete Mathematics and Its Applications. New York, NY: McGraw-Hill Education.

[5] Sturtevant, N. R. (2012). Benchmarks for grid-based pathfinding. IEEE Transactions on Computational Intelligence and AI in Games, 4(2), 144-148.

[6] Weisstein, Eric W. All-Pairs Shortest Path. MathWorld: A Wolfram Web Resource. http://mathworld.wolfram.com/All-PairsShortestPath.html, diakses 15 Juli 2019.

[7] Cui, X., & Shi, H. (2011). A*-based pathfinding in modern computer games. International Journal of Computer Science and Network Security, 11(1), 125-130.

[8] Patel, Amit (2004). Heuristics for grid maps. Amit's A* Pages. http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html #heuristics-for-grid-maps. Diakses 19 Agustus 2019.